Improved Algorithms for Min Cut and Max Flow in Undirected Planar Graphs

Giuseppe F. Italiano*
Dipartimento di Informatica, Sistemi e Produzione
University Rome "Tor Vergata"
Rome, Italy
italiano@disp.uniroma2.it

Yahav Nussbaum†
The Blavatnik School of Computer Science
Tel Aviv University
Tel Aviv, Israel
yahav.nussbaum@cs.tau.ac.il

Piotr Sankowski‡
Institute of Informatics
University of Warsaw
Warsaw, Poland
sank@mimuw.edu.pl

Christian Wulff-Nilsen§
School of Computer Science
Carleton University
Ottawa, Canada
koolooz@diku.dk

ABSTRACT

We study the min st-cut and max st-flow problems in planar graphs, both in static and in dynamic settings. First, we present an algorithm that given an undirected planar graph and two vertices s and t computes a min st-cut in $O(n \log \log n)$ time. Second, we show how to achieve the same bound for the problem of computing a max st-flow in an undirected planar graph. These are the first algorithms breaking the $O(n \log n)$ barrier for those two problems, which has been standing for more than 25 years. Third, we present a fully dynamic algorithm maintaining the value of the min st-cuts and the max st-flows in an undirected plane graph (i.e., a planar graph with a fixed embedding): our algorithm is able to insert and delete edges and answer queries for min st-cut/max st-flow values between any pair of vertices s and t in $O(n^{2/3} \log^{8/3} n)$ time per operation. This result is based on a new dynamic shortest path algorithm for planar graphs which may be of independent interest. We remark that this is the first known non-trivial dynamic algorithm for min st-cut and max st-flow.

*Work partially supported by the EU (Network of Excellence "EuroNF: Anticipating the Network of the Future - From Theory to Design") and by the Italian MIUR Project AlgoDEEP.

†Work partially supported by the United States - Israel Binational Science Foundation, Grant number 2006204.

‡Work partially supported by the Polish Ministry of Science, Grant N N206 355636 and by the ERC StG Project PAAl no. 259515.

§Work partially supported by NSERC and MRI.

STOC'11, June 6–8, 2011, San Jose, California, USA.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms

General Terms

Algorithms,Theory

Keywords

Planar graph, min cut, max flow, dynamic algorithm

1. INTRODUCTION

The min cut and max flow problems have been at the heart of algorithmic research on graphs for over 50 years. Classical algorithms include Ford-Fulkerson, Edmonds-Karp, and the push-relabel algorithms of Goldberg and Tarjan. The latter algorithm with dynamic trees runs in $O(mn \log(n^2/m))$ time, where m is the number of edges and n is the number of vertices [13]. For sparse graphs, i.e., for $m = O(n)$, this is the fastest known algorithm for max flow.

Particular attention has been given to solving those problems on planar graphs, not only because they often admit faster algorithms than general graphs but also since planar graphs arise naturally in many applications. The pioneering work of Ford and Fulkerson [8, 9], which introduced the max flow/min cut theorem, also contained an elegant algorithm for computing max st-flows in (s,t)-*planar* graphs (i.e., planar graphs where both the source s and the sink t lie on the same face). The algorithm was implemented to work in $O(n \log n)$ time by Itai and Shiloach [17]. Later, a simpler algorithm for the same problem was given by Hassin [14], who reduced the problem to a single-source shortest path computation in the dual graph. Henzinger et al. [16] showed that single-source shortest paths in planar graphs can be found in $O(n)$ time. As a result, a min st-cut and a max st-flow can be found in $O(n)$ time in (s,t)-planar graphs.

Itai and Shiloach [17] generalized the approach to the case of general undirected planar (i.e., not only (s,t)-planar) graphs, by observing that a min st-cut separating vertices s and t in a planar graph G is related to the min weight cycle

that separates faces f_s and f_t (corresponding to vertices s and t) in the dual graph. The resulting algorithm makes $O(n)$ calls to a shortets path algorithm and thus runs in a total of $O(n^2 \log n)$ time. Reif [24] improved this bound by finding the minimum weight separating cycle with a divide-and-conquer approach using only $O(\log n)$ runs of the (s,t)-planar algorithm: this yields an $O(n \log^2 n)$ time algorithm to compute a min st-cut for undirected planar graphs. Later, Frederickson [12] improved the running time of Reif's algorithm to $O(n \log n)$. The same result can be obtained by using the $O(n)$ time shortest path algorithm in [16]. Hassin and Johnson [15] extended the min st-cut algorithm of Reif to compute a max st-flow in $O(n \log n)$ time as well. In summary, the best bound known for computing min st-cuts and max st-flows in planar undirected graphs is $O(n \log n)$.

The first contribution of this paper improves the time bound for computing min st-cuts in planar undirected graphs to $O(n \log \log n)$. In order to achieve this bound, we do not depart from Reif's approach [24]. Instead we speed it up using a two-phase approach. The first phase runs a "coarse" version of Reif's algorithm which only determines a subset of the min st-cut candidates found by the original algorithm. We obtain a running time of $O(n \log \log n)$ for this phase using the fast Dijkstra variant of Fakcharoenphol and Rao [7]. In the second phase, the remaining min st-cut candidates are found exactly as in the algorithm by Reif but since the first phase partitions the problem into simpler subproblems, we can show that the second phase also runs in $O(n \log \log n)$ time.

As our second contribution, we show that a max st-flow in undirected planar graphs can be computed within the same $O(n \log \log n)$ time bound. This is a consequence of our new min st-cut algorithm together with the algorithm of Hassin and Johnson [15]. The algorithms presented in this paper are the first algorithms that break the $O(n \log n)$ long-standing barrier for min st-cut and max st-flow problems in undirected planar graphs. Computing min st-cuts in undirected planar graphs is the bottleneck in algorithms for other problems, including global min cut in planar undirected graphs and min st-cuts and shortest non-trivial cycles in undirected graphs embedded on surfaces of bounded genus. Our result thus improves the time bound for these problems as well.

As our third contribution, we present a fully dynamic data structure for maintaining information about min st-cuts and max st-flows in an undirected *plane* graph (i.e., a planar graph with a fixed embedding): our algorithm is able to insert and delete edges and report the value of a min st-cut/max st-flow for any pair of vertices s and t in $O(n^{2/3} \log^{8/3} n)$ time per operation. This result is based on our new techniques for the static min st-cut algorithm and on a new dynamic shortest path algorithm for planar graphs which may be of independent interest. We remark that this is the first known non-trivial algorithm for the min st-cut and max st-flow problems in a dynamic setting.

2. PRELIMINARIES

For a graph $G = (V, E)$, define a *piece* P of G to be the subgraph of G induced by a subset of E. In G, vertices of P incident to vertices not in P are the *boundary vertices* of P and we denote the set of them by ∂P. All other vertices of P are *interior vertices* of P. We extend this notation and use ∂H to denote the set of all boundary vertices of a known division of G in a subgraph H.

We will identify an st-cut with the set of edges from the s-side to the t-side of the cut.

2.1 r-Division

Frederickson [10] showed how to obtain, for any parameter $r \in (0, n)$, an *r-division* of a planar graph G, which is a division of (the edges of) G into $O(n/r)$ pieces each containing $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices. He gave an $O(n \log r + (n/\sqrt{r}) \log n)$ time algorithm to find such a division. We will need a stronger result. More precisely, assuming G is plane, define the *holes* of a piece P to be bounded faces of P which are not faces of G. Then we have the following result.

THEOREM 1. *For a plane n-vertex graph, an r-division in which each piece has $O(1)$ holes can be found in $O(n \log r + (n/\sqrt{r}) \log n)$ time.*

PROOF. We will only give a sketch of the proof here. For more details, see Appendix A and the original paper by Frederickson [10].

The idea is to first find a spanning forest, in which each tree has $\Theta(\sqrt{r})$ size, and to contract the graph on these trees. Let G' be the resulting graph. It has $n' = O(n/\sqrt{r})$ vertices. Now, find an r-division of G' using a simple recursive algorithm with $O(n' \log n') = O((n/\sqrt{r}) \log n)$ running time. This r-division has $O(n'/r) = O(n/r^{3/2})$ pieces each of size r and there are $O(n'/\sqrt{r}) = O(n/r)$ boundary vertices in total.

Expand G' back to G. In G, there are now $O(n/r)$ pieces of size $O(\sqrt{r})$ resulting from expanded boundary vertices and $O(n/r^{3/2})$ pieces of size $O(r^{3/2})$ resulting from expanded interior vertices of G'. Now, find an r-division inside each piece of size $O(r^{3/2})$. This takes $O((n/r^{3/2}) r^{3/2} \log r) = O(n \log r)$ time. The result is an r-division of G and the time to find it is $O(n \log r + (n/\sqrt{r}) \log n)$.

The above procedure of Frederickson does not ensure $O(1)$ holes in each piece. We modify the procedure as follows. For the $O(n \log n)$ time recursive algorithm (see Lemmas 1 and 2 in [10]), we apply Miller's cycle separator theorem [23] instead of the separator theorem of Lipton and Tarjan [22]. At each recursive step, we find a cycle that splits the current piece P into two subpieces P_1 and P_1'. Let h be the number of holes of P. One of the two subpieces, say P_1', may have $h+1$ holes. We split P_1' in two using an idea of Fakcharoenphol and Rao [7]: contract all its holes into super-vertices and apply Miller's theorem but with vertex weights evenly distributed on the super-vertices. In the resulting two subpieces P_2 and P_3 of P_1', expand the holes back. Then as shown in [7], P_1, P_2, and P_3 each have at most h holes.

It follows easily that plugging this idea into Frederickson's r-division algorithm gives an r-division where each piece has $O(1)$ holes. □

Throughout the paper, when we talk about an r-division, we assume it has the form in Theorem 1.

2.2 Dense Distance Graphs

If G is edge-weighted, we define the *dense distance graph* of a piece P to be the complete graph on ∂P where each edge (u, v) has weight equal to the shortest path distance (w.r.t. the edge weights) in P between u and v. In order to compute dense distance graphs for all pieces we use Klein's algorithm [20] which for a plane graph H with p vertices

and a fixed face f can report any shortest path distance $d_H(u,v)$, where either u or v is on f, in $O(\log p)$ time after $O(p \log p)$ time for preprocessing[1]. Since a piece P has $O(\sqrt{r})$ boundary vertices and $O(1)$ holes, applying Klein's algorithm to each face of P containing boundary vertices gives the dense distance graph of P in $O(r \log r)$ time. Since there are $O(n/r)$ pieces, we get the following result.

LEMMA 2. *The dense distance graphs of all pieces in an r-division can be computed in $O(n \log r)$ time.*

2.3 Fast Dijkstra

The dense distance graphs can be used to speed up shortest path computations using Dijkstra's algorithm. It was shown by Fakcharoenphol and Rao ([7], Section 3.2.2) that a Dijkstra-like algorithm can be executed on a dense distance graph of a piece P in $O(|\partial P| \log^2 |P|)$ time. Having constructed the dense distance graphs, we can run Dijkstra in time almost proportional to the number of vertices (rather than to the number of edges, as in standard Dijkstra). For this, we require that the underlying planar graph has constant degree which we can assume without loss of generality.

LEMMA 3. *Dijkstra's algorithm can be run in $O(b \log^2 n)$ time on a graph composed of dense distance graphs with a total of b boundary vertices (counted with multiplicity).*

PROOF. We use the data structure of Fakcharoenphol and Rao [7] for each dense distance graph. Minimum distance vertices from each of the $O(b)$ dense distance graphs are kept in a global heap. □

3. REIF'S ALGORITHM

Let G^* be a plane undirected connected graph, which we refer to as the *primal graph*. We define a plane undirected *dual graph* $G = (V, E, w)$ as follows. Each face of G^* corresponds to a vertex in G and for each edge e^* in G^* there is a dual edge e in G connecting the vertices corresponding to the two faces of G^* incident to e^*. In general, G is a multigraph. The weight of e in G is the same as the weight of e^* in G^*. Throughout the paper we will refer to vertices of the dual graph G interchangeably as (dual) vertices or faces.

Let s and t be any two vertices of G^*. We consider the problem of finding a min st-cut in G^*. There is a well-know duality between cuts in G^* and cycles in G. For the two faces s and t in G, an *st-separating cycle* in G is a simple cycle containing one of the faces s and t in its interior and the other in its exterior. The following lemma was proven by Itai and Shiloach [17].

LEMMA 4. *A min st-separating cycle in G defines a min st-cut in G^*.*

Reif's algorithm makes use of this duality. First, let us make the simplifying assumption that G is simple. If not, we can always subdivide edges by adding degree-two vertices and this will not change the asymptotic complexity of the problem. The algorithm starts by computing a shortest path $\pi = p_1 \to p_2 \to \cdots \to p_{|\pi|}$ from an arbitrary vertex p_1 on face s to an arbitrary vertex $p_{|\pi|}$ on face t in G. Then an *incision* in G along π is made as follows. Remove the set

[1]It is assumed in [20] that f is the external face but it can be generalized to any face by reembedding H.

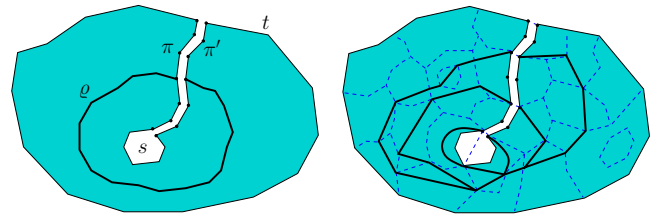

Figure 1: (a): In cut-open graph G_{st}, Reif's algorithm computes a shortest path ϱ from the midpoint on π to the midpoint on π' and recurses on the two subgraphs generated. (b): The coarse version of Reif's algorithm only computes shortest paths between boundary vertices on the cut-path. A refined version is then applied to find the remaining shortest paths. Only shortest paths from the coarse version are shown. Dashed line segments show the boundaries of pieces in the r-division.

E_r of edges emanating right of π in the direction from s to t. Insert a copy $\pi' = p'_1 \to p'_2 \to \cdots \to p'_{|\pi|}$ of π and for each edge $(p_i, u) \in E_r$, add edge (p'_i, u). We let G_{st} be the resulting graph, see Figure 1(a).

Next, Reif's algorithm computes a shortest path ϱ in G_{st} from the midpoint $p_{\lceil |\pi|/2 \rceil}$ of π to the midpoint $p'_{\lceil |\pi|/2 \rceil}$ of π'. This splits G_{st} into two subgraphs and splits π and π' into two halves, one for each side of ϱ. In each of the two subgraphs, degree two-vertices are removed by merging their incident edges. This is done to limit the size of the subgraphs generated. The algorithm then recurses on the two subgraphs and the two subpaths.

Let ϱ_i be the shortest path found by the algorithm and let p_i and p'_i be the first and last vertex of ϱ_i, respectively. Then the cycle in G obtained from ϱ_i by identifying p_i with p'_i is a min st-separating cycle in G. By Lemma 4, this cycle defines a min st-cut in G^*.

With Dijkstra's shortest path algorithm, Reif's algorithm runs in $O(n \log^2 n)$ time. This can be improved to $O(n \log n)$ time with Frederickson's algorithm [10] or by speeding up Reif's algorithm using the linear time shortest path algorithm of Henzinger et al. [16]. In the next section, we will further improve Reif's algorithm to get $O(n \log \log n)$ running time.

4. FASTER MIN st-CUT ALGORITHM

In this section, we present our min st-cut algorithm and give the claimed $O(n \log \log n)$ time bound. To ease the presentation, we leave out some details of the algorithm and return to them in Section 4.3. We start with the following simple lemma.

LEMMA 5. *Let s and t be faces in a planar undirected n-vertex graph G and let π be a given shortest path between a vertex on s and a vertex on t. In an application of Reif's algorithm to find a min st-separating cycle in G, a subproblem defined by a subgraph H and a subpath of π of length $O(\log^c n)$ for a constant c can be solved in $O(|H| \log \log n)$ time.*

PROOF. Recursion depth for Reif's algorithm in H is only $O(\log(\log^c n)) = O(\log \log n)$ so the running time for the

subproblem is $O(|H| \log\log n)$ using the shortest path algorithm in [16]. $\square$

We essentially run Reif's algorithm but speed part of it up with the Dijkstra variant given in Lemma 3. Recall that G denotes the dual of a plane embedding of the input graph. We need to find a min st-separating cycle in G, where s and t are faces.

Let π be a shortest path in G from an arbitrary vertex on s to an arbitrary vertex on t. We can find this path in linear time using the algorithm in [16]. We first run a "coarse" version of Reif. This will identify in $O(n \log\log n)$ time a subset of all the st-separating cycles found by the original algorithm. More precisely, the set of cycles found will split G into subgraphs each of which contains a subpath of π of length $O(\log^c n)$ for some constant c. We then run the "refined" Reif algorithm by applying Lemma 5 to each subgraph and its associated subpath of π. This will find the min st-separating cycle in G. By ensuring that the total size of the subgraphs is $O(n)$, the entire algorithm runs in $O(n \log\log n)$ time. Figure 1(b) illustrates the output of the first phase of our algorithm.

4.1 First Phase

We will now describe the first phase of our algorithm which is the coarse version of Reif's algorithm.

r-Division.

We apply Theorem 1 to obtain an r-division of G for $r = \log^6 n$. This takes $O(n \log r + (n/\sqrt{r}\,) \log n) = O(n \log\log n)$ time.

Cutting Pieces Open.

We make an incision in G along the shortest path π as in Reif's algorithm. This induces incisions in those pieces containing parts of π and we update the pieces accordingly. If a boundary vertex of a piece belongs to π before the incision, we define both of its two copies after the incision as boundary vertices of that piece. Note that there will still be only $O(\sqrt{r}\,)$ boundary vertices in each piece and these boundary vertices will still be on a constant number of holes after the incision. Hence, the resulting set of pieces forms an r-division in the cut-open graph.

Dense Distance Graphs.

We find in $O(n \log r) = O(n \log\log n)$ time dense distance graphs for the pieces in the r-division using Lemma 2. The edge weights of each dense distance graph are represented in a matrix with $O(\sqrt{r}\,)$ rows and columns.

The total number of boundary vertices of these pieces is $O(n/\sqrt{r}\,)$ and it follows from Lemma 3 that a shortest path between any two boundary vertices in the r-division can be computed in $O((n/\sqrt{r}\,)\log^2 n) = O(n/\log n)$ time. Note that this shortest path consists of edges from the dense distance graphs so it is an implicit representation of a shortest path in the underlying cut-open graph.

Coarse Reif.

For the sequence of vertices of π, consider the (possibly empty) subsequence of vertices that are boundary vertices in pieces of the r-division. These vertices partition π into subpaths each of which is contained in a piece. The coarse version of Reif's algorithm is the normal algorithm of Reif restricted to this subsequence and using the $O(n/\log n)$ time shortest path algorithm for each vertex in this subsequence. As in Reif's algorithm, for each shortest path ϱ computed, the graph is split into two subgraphs along ϱ and we recurse on each of them.

We need the subgraphs generated to have in total $O(n/\sqrt{r}\,)$ boundary vertices and we do this by ensuring that they do not share too many boundary vertices. We deal with this in Section 4.3. Since recursion depth is $O(\log n)$, total time for the first phase of our algorithm is $O(n)$ in addition to the $O(n \log\log n)$ time to find the r-division and to set up the dense distance graphs. The st-separating cycles found in this phase partition G into subgraphs each containing a subpath of π fully contained in a piece of the r-division. Hence, the length of each such subpath is bounded by the size $O(r) = O(\log^6 n)$ of a piece.

Subgraphs for Recursive Calls.

When applying the coarse version of Reif's algorithm, we need to find the subgraphs for recursive calls. Consider a subgraph H in some recursive call. We associate with H the boundary vertices belonging to H and the cyclic orderings of these vertices on holes and external faces of pieces. Whenever we need a distance $d_{P\cap H}(u,v)$, where u and v are boundary vertices of a piece P, we make a look-up in the dense distance graph for P in G. It is easy to see that this will give $d_{P\cap H}(u,v)$ if u and v are connected in $P \cap H$. Otherwise, $d_{P\cap H}(u,v)$ is infinite. Before running the fast Dijkstra variant for H we can partition in $O(|\partial P \cap H|)$ time the set $\partial P \cap H$ into groups induced by the connected components in $P \cap H$. With this information, we can report $d_{P\cap H}(u,v)$ in constant time for all $u, v \in \partial P \cap H$.

It follows that the total time to find a shortest path in H is $O(h \log^2 n)$, where h is the number of boundary vertices in H. Over all such subgraphs, this will be $O(n)$ time.

4.2 Second Phase

In order to run the second phase of our algorithm, we need to convert the shortest paths consisting of edges from dense distance graphs to the underlying shortest paths in G and we need to find the subgraphs of G bounded by these paths. In the next subsection we show how to do this in $O(n)$ time such that the total size of the subgraphs is $O(n)$. Applying Lemma 5 with constant $c = 6$ to each subgraph, we get $O(n \log\log n)$ time for the second phase of our algorithm. Hence, the entire algorithm has $O(n \log\log n)$ running time.

4.3 Overlapping Subgraphs

We need to ensure that the total number of boundary vertices in the subgraphs generated by the coarse version of Reif's algorithm is $O(n/\sqrt{r}\,)$. The problem is that subgraphs overlap so a boundary vertex can belong to several subgraphs. The original algorithm of Reif ensures linear total size by deleting, in every subgraph generated, each degree-two vertex by replacing the two edges e_1 and e_2 incident to it by one whose weight is the sum of the weights of e_1 and e_2.

First Phase.

In the first phase of our algorithm, we do something similar. Let H be a subgraph in the coarse version of Reif's algorithm. We assume that H is bounded by two shortest paths in the cut-open graph, π_1 starting from a boundary

vertex p_1 on π and π_2 starting from a boundary vertex p_2 on π; the case where H is bounded by only one shortest path is handled similarly. Here we regard π_1 and π_2 as paths consisting of edges in the dense distance graphs of pieces.

The algorithm partitions H into two subgraphs H_1 and H_2 with a shortest path π_3 in H from a boundary vertex p_3 between p_1 and p_2 on π. For $i = 1, 2$, assume that H_i contains π_i and let $\pi'_i = \pi_i \cap \pi_3$. We may pick π_3 such that π'_i is a (possibly empty) path (this will always be the case if ties in distances are broken consistently by the algorithm in Lemma 3). Suppose π'_i contains at least three (boundary) vertices and let q_i and q'_i be the endpoints of this path. Note that in the underlying graph G, each interior vertex of π'_i has degree 2 in H_i so for this subgraph, we do not include π'_i but instead a super edge from q_i to q'_i of weight equal to that of π'_i.

Clearly, the above strategy ensures that distances are preserved. To bound the size of subgraphs, note that all interior vertices of π'_1 and π'_2 are in only one of the subgraphs H_1 and H_2 and all vertices of $\pi_3 \setminus (\pi'_1 \cup \pi'_2)$ are in H_1 and H_2 and in no other subgraph. Furthermore, paths π'_1 and π'_2 together have at most four endpoints. It follows that the total size of all the subgraphs generated by Reif is bounded by a constant times the total number of boundary vertices plus the number of times a subgraph is split. In the first phase we apply Reif to $O(n/\sqrt{r})$ vertices, so the graph is split the same number of times. Hence, the total size of all graphs created is $O(n/\sqrt{r})$, as desired. We apply Lemma 3 and regard each super edge as an additional dense distance graph with one edge. Since the total number of super edges is bounded by the number of subgraphs, there are $O(n/\sqrt{r})$ boundary vertices in total so the time to compute shortest paths in the first phase is $O((n/\sqrt{r})\log^3 n) = O(n)$.

Second Phase.

We also face the problem with overlapping subgraphs when converting the shortest paths consisting of dense distance graph edges to shortest paths in G for the second phase of the algorithm. Since G has constant degree, Klein's algorithm [20] can report the underlying path in G corresponding to a dense distance graph edge in time proportional to the length of the path. Hence, after the first phase we can obtain each of the shortest paths computed in time proportional to their total size. However, this size can be superlinear since the paths may share many vertices. We deal with this problem in the following. We will show that an implicit representation of the paths can be computed in $O(n)$ time.

Implicit Representation of Paths.

Define $p_{i_1}, \ldots, p_{i_k}$ as the ordered sequence of vertices of π from which the coarse Reif algorithm has computed shortest paths $\pi_{i_1}, \ldots, \pi_{i_k}$. Let $p'_{i_1}, \ldots, p'_{i_k}$ be the other endpoints of these paths. We start by obtaining the shortest path ϱ_{i_1} in G from p_{i_1} to p'_{i_1} using Klein's algorithm on each dense distance graph edge of π_{i_1}. This takes $O(|\varrho_{i_1}|)$ time.

To find the shortest path ϱ_{i_2} in G from p_{i_2} to p'_{i_2}, we similarly apply Klein's algorithm on π_{i_2}. If we encounter no vertices already visited, we obtain the entire path and move on to π_{i_3}. Otherwise, let v_1 be the first already visited vertex and let ϱ_{v_1} be the path found. We stop the algorithm when reaching v_1 and instead start obtaining vertices of ϱ_{i_2} backwards from p'_{i_2} until reaching an already visited vertex v_2. Let ϱ_{v_2} be the path found but ordered from v_2 to p'_{i_2}.

If v_2 is on ϱ_{i_1}, a simple property of shortest paths allows us to choose ϱ_{i_2} as the concatenation of ϱ_{v_1}, the subpath of ϱ_{i_1} from v_1 to v_2, and ϱ_{v_2}, in that order. This gives us an implicit representation of ϱ_{i_2} in $O(|\varrho_{i_2} \setminus \varrho_{i_1}|)$ time.

Let (u, v) be a super edge or an edge in a dense distance graph. This edge need not represent the same underlying shortest path as the edge (v, u) since shortest paths need not be unique. Hence, it may happen that v_2 is on ϱ_{v_1} and not on ϱ_{i_1}. If so, we can redefine ϱ_{i_2} to be the subpath of ϱ_{v_1} from p_{i_2} to v_2 followed by ϱ_{v_2}. Letting ϱ_{v_2,v_1} be the subpath of ϱ_{v_1} from v_2 to v_1, total time to find ϱ_{i_2} is $O(|\varrho_{i_2}| + |\varrho_{v_2,v_1}|) = O(|\varrho_{i_2} \setminus \varrho_{i_1}| + |\varrho_{v_2,v_1}|)$. Since none of the vertices on ϱ_{v_2,v_1}, excluding v_2, will be visited again, we can afford to spend time $O(|\varrho_{v_2,v_1}|)$.

Repeating this process for the remaining shortest paths $\pi_{i_3}, \ldots, \pi_{i_k}$ gives an implicit representation of the corresponding shortest paths in G and it follows from the above analysis that running time is $O(n)$. In linear time it is then easy to obtain from this representation the desired subgraphs needed in the second phase of our algorithm and to ensure that they have total linear size.

4.4 Global Min Cuts and Min Cuts on Surfaces

Computing a min st-cut in a planar graph is the bottleneck in algorithms for some other problems. Our algorithm improves the best known time bound for such problems.

Chalermoosk et al. [3] showed a simple algorithm for finding a *global min cut* in an undirected planar graph, i.e., a min st-cut of smallest weight over all distinct pairs of vertices s and t. Their algorithm actually finds the *girth*, i.e., a shortest cycle, in the dual graph. The algorithm works in a divide-and-conquer fashion, where in each step the graph is divided into two subgraphs, and a single min st-cut is computed. Using our algorithm we improve the time bound of [3].

THEOREM 6. *The global min cut and girth of an undirected planar graph can be computed in $O(n \log n \log\log n)$ time.*

For the unweighted case, Weimann and Yuster [25] presented an $O(n \log n)$ time algorithm for these problems.

We show that our algorithm improves also a result of Kutz [21] which is the basis of some algorithms for graphs embedded on a surface of bounded genus. We refer the reader to [21] for more details and for the relevant definitions. Consider a graph embedded in a plane with two *boundaries*. We can use Reif's algorithm to find a shortest cycle that separates the two boundaries. We simply make one of the boundaries the face s and the other the face t. Kutz [21] used this tool as a building block for a solution of the following problem: Given a graph G embedded on an orientable surface of bounded genus g and a *shortest system of loops*, find a shortest cycle which crosses the set of loops k times in a specified order (the same loop can be crossed multiple times). The idea of the solution for this problem is to take k copies of the *fundamental domain* defined by the system of loops, glue them according to the sequence of loop crossings, and identify the first and the last copy. The result of the gluing process is a planar graph with two boundaries, in which we find the shortest cycle that separates the two boundaries using Reif's algorithm – this is the required shortest cycle. Using our algorithm, we can

find a shortest cycle with a given sequence of crossings in $O(kn \log\log kn)$ time, where $O(kn)$ is the size of k copies of the graph G.

Kutz [21] showed how to use this procedure $O(g^{O(g)})$ times, with crossing sequences of length $O(g)$, to find a shortest *non-contractible cycle* and a shortest *non-separating cycle* in G. Chambers el al. [4] used this procedure in a similar way to find a min st-cut in G. If $g = O(n^{1-\varepsilon})$ we can find a shortest system of loops in $O(n)$ time by using the algorithm of Henzinger et al. [16] instead of Dijkstra's algorithm in the greedy algorithm of Erickson and Whittlesey [6]. Our improvement of Reif's algorithm therefore leads to the following improvements in the algorithms of [21] and [4]:

THEOREM 7. *For an undirected graph embedded on an orientable surface of genus g, a shortest non-contractible cycle and a shortest non-separating cycle can be found in $O(g^{O(g)} n \log\log n)$ time.*

THEOREM 8. *For an undirected graph embedded on an orientable surface of genus g, a min st-cut can be found in $O(g^{O(g)} n \log\log n)$ time.*

Our results improve the time bounds for these problems for small genus g. For larger g, the $O(g^3 n \log n)$ time algorithm of Cabello and Chambers [2] for non-contractible and non-separating shortest cycles, and the $O(2^{O(g)} n \log n)$ time algorithm of Erickson and Nayyeri [5] for min st-cuts have better time bounds.

5. FASTER MAX st-FLOW ALGORITHM

Hassin and Johnson [15] extend Reif's algorithm and show how to find a max st-flow using the value of a min st-cut and the shortest paths $\varrho_1 \dots \varrho_{|\pi|}$ between the vertices $p_1, \dots, p_{|\pi|}$ and $p'_1, \dots, p'_{|\pi|}$ of the two copies of the path π, which Reif's algorithm has found. The running time that Hassin and Johnson state in [15, Theorem 2] for their max st-flow algorithm is $O(n \log n)$ using Dijkstra's algorithm or $O(n\sqrt{\log n})$ using Frederickson's shortest path algorithm [12]. With the more recent shortest path algorithm of Henzinger et al. [16] the running time of the algorithm becomes $O(n)$.

We use the algorithm of Hassin and Johnson [15] without a change. We have already found the value of a min st-cut but we also need to implicitly represent all the shortest paths ϱ_i. In Section 4.3, we found an implicit representation of the shortest paths that we computed in the coarse Reif phase. For the second phase, we replace the use of Reif's algorithm in Section 4.2 with the implementation of Hassin and Johnson [15, Section 3], which we apply to each subgraph. The only difference between Reif's algorithm and the one of Hassin and Johnson is that the latter keeps an implicit representation of $O(n)$ size of all the shortest paths that it finds, so the running time of our algorithm which is obtained by Lemma 5 remains $O(n \log\log n)$. The algorithm of Hassin and Johnson keeps a representation of the shortest paths ϱ_i using super edges similar to the way we do in Section 4.3. The size of this representation is linear in the size of the input graph, and since the total size of all the subgraphs that we have in the second phase is $O(n)$, we get the desired representation in $O(n)$ space.

The running time of our min st-cut algorithm remains $O(n \log\log n)$ and we run the algorithm of [15] on its output. The latter algorithm takes $O(n)$ time using the shortest path algorithm of [16] and we have shown the following.

THEOREM 9. *A max st-flow in an undirected planar graph can be computed in $O(n \log\log n)$ time.*

6. DYNAMIC MAX st-FLOW VALUES

In this section, we consider the problem of computing shortest path distances and min st-cut/max st-flow values in a dynamic environment. Most of the ideas presented here are not entirely new but combined they give a new algorithm for this problem that is simpler and improves on previously known approaches. We first show how to maintain a planar graph G with positive edge weights under an intermixed sequence of the following operations: *insert*(x, y, c) add to G an edge of weight c between vertex x and vertex y, provided that the embedding of G does not change; *delete*(x, y) delete from G the edge between vertex x and vertex y; *distance*(x, y) return the distance in G from vertex x to vertex y.

Klein [20] gave a similar data structure which does not allow edge insertion or deletion but which supports weight updates, with $O(n^{2/3} \log^{5/3} n)$ update and query time. We extend this data structure to include the operations *insert* and *delete*.

In our dynamic algorithm we maintain an r-division of G together with dense distance graphs of all its pieces (we specify r below). This information will be recomputed every order $\sqrt{r}$ operations. We now show how to handle the different operations. To insert resp. delete an edge (x, y) when x and y are in a single piece P, we add resp. delete the edge from P, and recompute the dense distance graph of this piece. This can be carried out in $O(r \log r)$ time as shown in Section 2.2. When we insert an edge (x, y) such that x and y are not in a common piece, then either x is a boundary vertex of every piece that contains it, or x is in a single piece P_x. In the latter case, the vertex x is not a boundary vertex of P_x, but it is adjacent to some face to which y is also adjacent. Hence, adding an edge between x and y will make x a boundary vertex of P_x without increasing the number of holes. After we have made x a boundary vertex, we recompute the dense distance graph of P_x. This also requires $O(r \log r)$ time. If y is not a boundary vertex, we process the single piece that contains it similarly. We treat the edge (x, y) as a new piece containing only this edge. Since an r-division is recomputed every order $\sqrt{r}$ operations, this will guarantee that throughout the sequence of operations each piece will always have at most $O(r)$ edges and $O(\sqrt{r}\,)$ boundary vertices. Amortizing the initialization over order $\sqrt{r}$ operations yields $O(\frac{n \log r + \frac{n}{\sqrt{r}} \log n}{\sqrt{r}} + r \log r)$ time per update.

To answer the query *distance*(x, y), where x is in a piece P_x and y is in a piece P_y, we compute shortest paths from x to the boundary vertices of P_x inside P_x, then continue with the fast Dijkstra variant on the boundary vertices, and finally compute shortest paths from the boundary vertices of P_y to y inside P_y. This gives us a shortest path from x to y if they are in different pieces. If $P_x = P_y$ then, in addition to the above, we also compute a shortest path between x and y inside the piece, and take this path if it is shorter than the previous one. By Lemma 3, we can compute a shortest path in the entire graph between two boundary vertices in $O((n/\sqrt{r}\,) \log^2 n)$ time. Computing shortest paths inside a piece takes $O(r)$ time using the algorithm of Henzinger et

al. [16]. We conclude that we can answer a distance query in $O(r + (n/\sqrt{r})\log^2 n)$ time.

Setting $r = n^{2/3}/\log^{2/3} n$ gives the following lemma.

LEMMA 10. *Given a planar graph G with positive edge weights, we can insert edges, delete edges and report the distance between any pair of vertices in $O(n^{2/3}\log^{5/3} n)$ amortized time per operation.* [2]

6.1 Maintaining the Dual

Before turning our attention to the dynamic min st-cut problem, we need to show how to dynamically maintain the dual graph when the primal graph is updated. Given a planar graph $G^* = (V^*, E^*)$ we wish to perform an intermixed sequence of the following operations: *insert*(x, y, c) add to G^* an edge of weight c between vertex x and vertex y, provided that the embedding of G^* does not change; *delete*(x, y) delete from G^* the edge between vertex x and vertex y; *distance*(f_s, f_s) return a distance from face f_s to face f_t in G.

Things are now more involved as a single edge change in the primal graph G^* may cause more complicated changes in the dual graph G. In particular, inserting a new edge into the primal graph G^* results in splitting into two a vertex in the dual graph G, whereas deleting an edge in the primal graph G^* implies joining two vertices of G into one. As edges are inserted into or deleted from the primal graph, vertices in the dual graph are split or joined according to the embedding of their edges. To handle such splits and joins efficiently, we do as follows. Let f be a vertex of degree d in the dual graph: we maintain vertex f as a cycle C_f of d edges, each of cost 0. The actual edges originally incident to f, are made incident to one of the vertices in C_f in the exact order given by the embedding. It is now easy to see that in order to join two vertices f_1 and f_2, we need to cut their cycles C_{f_1} and C_{f_2}, and join them accordingly. This can be implemented in a constant number of edge insertions and deletions. Similarly, we can support vertex splitting with a constant number of edge insertions and deletions. Hence, the above set of operations can be supported within the same time bound as in Lemma 10.

6.2 Max st-Flow Queries

Given a planar graph $G^* = (V^*, E^*)$ we wish to perform an intermixed sequence of the following operations: *insert*(x, y, c) add to G^* an edge of weight c between vertex x and vertex y, provided that the embedding of G^* does not change; *delete*(x, y) delete from G^* the edge between vertex x and vertex y; *max-flow*(s, t) return the max-flow value from s to t in G^*.

Here, we need the following observation. This result is implied by [19].

COROLLARY 11. *Let π be a any path from an arbitrary vertex on face s to an arbitrary vertex on face t in G. Then Reif's divide-and-conquer algorithm can be executed on π.*

First, the modified algorithm computes for the middle vertex p_i of the path π in Corollary 11 a min st-separating cycle ϱ_i that passes through p_i. Next, it recurses on both sides of the cycle ϱ_i. The only difference is the way the cycle ϱ_i is computed. Instead of finding a shortest $p_i - p'_i$ path in G_{st} it looks for a shortest $p_i - p'_i$ path in G^2_{st}, where G^2_{st} is obtained by gluing together path π with π' and vice versa in two copies of G_{st}.

In order to simplify the presentation of our idea, we assume that pieces in the r-division do not contain holes. Hence, each piece has all its boundary vertices cyclically ordered on the external face. In Appendix B, we deal with the general case.

We define a *skeleton graph* $G_{\mathcal{S}} = (\partial G, E_{\mathcal{S}})$ to be a graph over the set of boundary vertices in the r-division of G. The edge set $E_{\mathcal{S}}$ is composed of infinite weight edges connecting consecutive (in the order on the hole) boundary vertices on each hole and the external face of each piece. By our holeless assumption, all boundary vertices in each piece lie on the external face of the piece, so the graph $G_{\mathcal{S}}$ is connected. We define a *patched graph* to be $\overline{G} = G \cup G_{\mathcal{S}}$. Note that this graph is still planar and the shortest distances do not change after adding infinite weight edges.

In the algorithm we maintain dynamically information about the distances in the dual graph G using the ideas from the previous section. In order to answer *max-flow* queries we use Corollary 11 and define path π as follows. Let b_s and b_t be any boundary vertices in pieces P_s and P_t respectively. Then π is the concatenation of a simple path from f_s to b_s in P_s, a simple path from b_s to b_t in $G_{\mathcal{S}}$, and a simple path from b_t to f_t in P_t.

Let us consider all edges inside P_s and P_t as individual pieces. There are only $O(r)$ of them. By our construction, all vertices on π are boundary vertices so our coarse version of Reif's algorithm in Section 4.1 will actually find the min st-cut and there is no need to run the second refined Reif algorithm. The execution time of the coarse algorithm for $r = n^{2/3}/\log^{2/3} n$ is $O(n^{2/3}\log^{8/3} n)$, as it essentially amounts to running the shortest path algorithm from the previous section order $\log n$ times. This finishes the details of the algorithm in the case where pieces have no holes. The modifications needed to prove the algorithm in the general case are given in Appendix B.

LEMMA 12. *Given a planar graph G with positive edge weights, each operation* insert, delete *and* max-flow *can be implemented in $O(n^{2/3}\log^{8/3} n)$ amortized time.* [3]

7. REFERENCES

[1] J. Alber, M. R. Fellows, and R. Niedermeier. Polynomial-time data reduction for dominating set. *J. ACM*, 51(3):363–384, 2004.

[2] S. Cabello and E. W. Chambers. Multiple source shortest paths in a genus g graph. In *SODA: Proc. 18th Annual ACM-SIAM Symposium*, pages 89–97, 2007.

[3] P. Chalermsook, J. Fakcharoenphol, and D. Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In *SODA: Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 828–829, 2004.

[4] E. W. Chambers, J. Erickson, and A. Nayyeri. Minimum cuts and shortest homologous cycles. In *SoCG: Proc. 25th ACM Symposium on Computational Geometry*, pages 377–385, 2009.

[2] The same worst-case time bounds can be obtained using a global rebuilding technique.

[3] Again, the same worst-case time bounds can be obtained using a global rebuilding technique.

[5] J. Erickson and A. Nayyeri. Minimum cuts and shortest non-separating cycles via homology covers. In *SODA: Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1166–1176, 2011.

[6] J. Erickson and K. Whittlesey. Greedy optimal homotopy and homology generators. In *SODA: Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1038–1046, 2005.

[7] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.

[8] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Can. J. Math.*, pages 399–404, 1956.

[9] L. R. Ford and D. R. Fulkerson. *Flows in Network*. Princeton Univ. Press, 1962.

[10] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs with applications. *SIAM J. Comput.*, 16:1004–1022, 1987.

[11] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[12] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[13] A. Goldberg and R. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[14] R. Hassin. Maximum flows in (s,t) planar networks. *IPL*, page 107, 1981.

[15] R. Hassin and D. B. Johnson. An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM J. Comput.*, 14(3):612–624, 1985.

[16] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.

[17] A. Itai and Y. Shiloach. Maximum flow in planar networks. *SIAM J. Comput.*, 8(2):135–150, 1979.

[18] D. B. Johnson. Parallel algorithms for minimum cuts and maximum flows in planar networks. *J. ACM*, 34(4):950–967, 1987.

[19] H. Kaplan and Y. Nussbaum. Minimum s-t cut in undirected planar graphs when the source and the sink are close. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, LIPIcs 9, pages 117–128, 2011.

[20] P. N. Klein. Multiple-source shortest paths in planar graphs. In *SODA '05: Proc. 16th Annual ACM-SIAM Symposium on Discrete algorithms*, pages 146–155, 2005.

[21] M. Kutz. Computing shortest non-trivial cycles on orientable surfaces of bounded genus in almost linear time. In *SoCG: Proc. 22nd ACM Symposium on Computational Geometry*, pages 430–438, 2006.

[22] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Applied Math.*, pages 177–189, 1979.

[23] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *J. Comput. System Sci.*, 32(3):265–279, 1986.

[24] J. Reif. Minimum *s*-*t* cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM J. Comput.*, 12:71–81, 1983.

[25] O. Weimann and R. Yuster. Computing the girth of a planar graph in $O(n \log n)$ time. *SIAM J. Discrete Math.*, 24(2):609–616, 2010.

APPENDIX

A. PROOF OF THEOREM 1

In this section, we prove Theorem 1, i.e., we show that an r-division can be found in $O(n \log r + (n/\sqrt{r}) \log n)$ time. We use an approach similar to that of Frederickson [10]: contract G, find an r-division of this smaller graph, expand the graph back to G, and split some of the resulting pieces further to get the desired r-division of the whole graph. First, however, we shall give a simple $O(n \log n)$ time algorithm. In Appendix A.3, this algorithm will be used to find an r-division of the contracted graph.

A.1 Weak r-Division

To obtain an r-division of G in $O(n \log n)$ time, we follow Frederickson's approach. First we find a *weak r-division* which is a division of G into $O(n/r)$ pieces each of size $O(r)$ and with a constant number of holes, such that the total number of boundary vertices over all pieces is $O(n/\sqrt{r})$. In Appendix A.2, we will then split pieces further to get the desired r-division in $O(n \log n)$ time.

Consider the following recursive algorithm to find a weak r-division: regard G as a piece with no boundary vertices, split it recursively into two subpieces with the cycle separator theorem of Miller [23] and recurse on them. The recursion stops when a piece has size at most r. As shown by Frederickson [10], this gives a weak r-division. However, it does not ensure a constant bound on the number of holes in each piece which we need in our application.

To deal with this, we use ideas of Fakcharoenphol and Rao [7] to keep the number of holes bounded by some constant h. The initial piece is the whole graph and thus contains no holes. Now, consider the general recursive step and let $P = (V_P, E_P)$ be the current piece. Assume it has at most h holes. Apply the cycle separator theorem to P with all vertices assigned weight $1/|V_P|$. This splits P into two subpieces $P_1 = (V_{P_1}, E_{P_1})$ and $P_1' = (V_{P_1'}, E_{P_1'})$, where $|V_{P_1}| = \alpha|V_P| + O(\sqrt{|V_P|})$ and $|V_{P_1'}| = (1-\alpha)|V_P| + O(\sqrt{|V_P|})$ for some $\frac{1}{3} \le \alpha \le \frac{2}{3}$. Assume w.l.o.g. that P_1 belongs to the interior of the separator cycle. Then this subpiece has at most h holes. However, since the separator cycle may have introduced a new hole in P_1', this subpiece may have $h+1$ holes.

Contract the holes of P_1' into *super vertices* and apply the cycle separator theorem with vertex weights distributed evenly on super vertices. Expand them back to holes and let P_2 and P_3 be the resulting two pieces. As shown in [7], the number of holes in each of the two subpieces will be a constant factor smaller than $h+1$ so if we pick h sufficiently large, P_2 and P_3 each have at most h holes. Now, we recurse on P_1, P_2, and P_3 until all pieces contain at most r vertices.

LEMMA 13. *The above procedure gives, for any parameter $r \in (0, n)$, a weak r-division of G where each piece has a constant number of holes. Running time is $O(n \log(n/r))$.*

The proof is similar to that of Lemma 1 in [10]; it will be given in the full version of this paper.

A.2 r-Division in $O(n \log n)$ Time

To obtain an r-division in $O(n \log n)$ time, we first find a weak r-division with Lemma 13. Each piece has $O(r)$ vertices and a constant number of holes but there may be more than order $\sqrt{r}$ boundary vertices in a single piece.

We continue to follow Frederickson's approach while ensuring a constant number of holes in each piece. If there is a piece P containing more than $c\sqrt{r}$ boundary vertices for some constant c, apply the cycle separator theorem as in the weak r-division procedure to obtain subpieces P_1 and P_1'. However, instead of distributing the vertex weights evenly on all vertices of P when applying the theorem, we now distribute weights evenly on boundary vertices only. We then infer subpieces P_2 and P_3 of P_1' as before by distributing vertex weights evenly on super vertices defined by contracted holes of P_1'. This is repeated until each piece has at most $c\sqrt{r}$ boundary vertices.

LEMMA 14. *The above procedure gives, for any parameter $r \in (0, n)$, an r-division of G in $O(n \log n)$ time.*

The proof is similar to that of Lemma 2 in [10] and it will be given in the full version of the paper.

A.3 A Faster Algorithm

We now show how to get the desired running time of $O(n \log r + (n/\sqrt{r}) \log n)$ in Theorem 1. We start by computing a spanning tree T of G (here, we assume that G is connected; we can always add infinite-weight edges to achieve this) and partitioning it into $\Theta(n/\sqrt{r})$ subtrees each of size $\Theta(\sqrt{r})$; the subtrees cover all vertices and are pairwise vertex-disjoint. This takes $O(n)$ time with the algorithm in [11].

Let G' be a plane multigraph obtained from G by contracting each subtree to a single vertex (G' inherits the embedding of G). This graph contains $O(n/\sqrt{r})$ vertices. To obtain the same asymptotic bound on the number of edges, we will turn G' into a so called thin graph. In a plane multigraph, a *bigon* is a face defined by two vertices and edges. A plane multigraph is *thin* if it contains no bigons. The following result from [1] shows that thin multigraphs are sparse.

LEMMA 15. *A thin n-vertex multigraph has $O(n)$ edges.*

Let G'' be the thin multigraph obtained from G' by identifying the two edges of a bigon with one edge and repeating this process until no bigons exist. We turn the graph G'' into a simple graph G''' by subdividing each edge (u, v) into two edges (u, w) and (w, v). Note that (u, v) corresponds to $O(\sqrt{r})$ edges in G' and in G; we subdivide each of them similarly such that the sum of weights of each edge pair equals the weight of the edge they subdivide. Now, G' is a simple graph and we color black those vertices of G' that correspond to contracted trees in G. All other vertices of G' are colored white.

By Lemma 15, G''' is a simple plane graph of size $O(n/\sqrt{r})$ so we can find an r-division of it in $O((n/\sqrt{r}) \log n)$ time with Lemma 14. This r-division consists of $O(n/r^{3/2})$ pieces each of size $O(r)$. We get an induced division of G' into pieces each consisting of $O(r)$ black vertices and $O(r^{3/2})$ white vertices. Furthermore, each piece in this division has $O(\sqrt{r})$ black boundary vertices and $O(r)$ white boundary vertices.

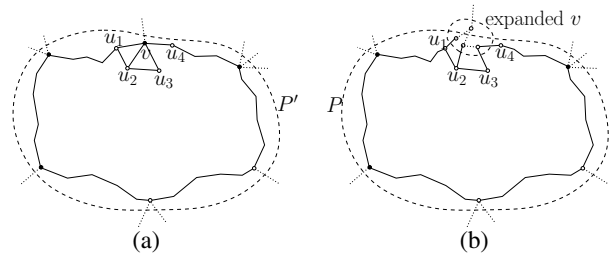

Figure 2: (a): A piece P' in the r-division of G' with a black boundary vertex v on the external face. (b): After expanding v to a tree when forming a corresponding piece $P \in \mathcal{P}_2$, new boundary vertices will also be on the external face. The same is true for holes. Only solid edges are part of the pieces.

Let $\mathcal{P}_1$ be the set of subtrees of G defined by the expanded black boundary vertices of pieces in the division of G'. Note that $|\mathcal{P}_1| = O(n/r)$ and each subgraph in $\mathcal{P}_1$ has size $O(\sqrt{r})$.

For each piece P in the division of G', let P' be the piece in G defined by the union of edges in P and subtrees from expanded black interior vertices of P. Let $\mathcal{P}_2$ denote the set of these pieces P'. Note that $|\mathcal{P}_2| = \Theta(n/r^{3/2})$ and each piece P' in $\mathcal{P}_2$ has size $O(r^{3/2})$. Furthermore, since there are $O(r)$ white boundary vertices in P' and each of the $O(\sqrt{r})$ black boundary vertices of the corresponding piece in G' contributes with at most $O(\sqrt{r})$ boundary vertices to P' when expanded, P' has $O(r)$ boundary vertices.

The pieces in $\mathcal{P}_1 \cup \mathcal{P}_2$ cover all edges in G and each edge is contained in exactly one piece. We will transform these pieces into an r-division of G.

First consider pieces $P \in \mathcal{P}_1$. The number of boundary vertices of P is bounded by the size $O(\sqrt{r})$ of P. Since P is a tree, all its boundary vertices are on the external face. Hence, P has no holes and we include it as part of the r-division of G.

Now, consider pieces $P \in \mathcal{P}_2$. The piece P' in the division of G' corresponding to P has a constant number of holes. We claim that the same holds for P. To show this, consider some black boundary vertex v in P' and let T_v be the tree in G obtained by expanding v. In P, v gets expanded into $O(\sqrt{r})$ boundary vertices all belonging to T_v. Since none of the edges of T_v belong to P by definition, these boundary vertices must all be on the same face of P; see Figure 2. Repeating this argument for all black boundary vertices of P', it follows that P has a constant number of holes.

Since P has size $O(r^{3/2})$ and $O(r)$ boundary vertices, we can find an r-division of it in $O(r^{3/2} \log r)$ time using Lemma 14 with a small modification: when finding a weak r-division of P, the boundary vertices and the holes of P will be regarded as boundary vertices and holes in the initial graph. The result will still be a weak r-division of P since the total number of boundary vertices will be $O(|P|/\sqrt{r} + r) = O(r) = O(|P|/\sqrt{r})$.

Total time to find r-divisions over all $P \in \mathcal{P}_2$ is $O(n \log r)$. Taking the union of the pieces in all these r-divisions together with the pieces in $\mathcal{P}_1$, we obtain the r-division of G in $O(n \log r + (n/\sqrt{r}) \log n)$ time. This proves Theorem 1.

B. PROOF OF LEMMA 12

When holes are present in the r-division the skeleton graph $G_{\mathcal{S}}$ may not be connected. Hence, the path connecting b_s and b_t cannot use boundary vertices only. Such a path can become too long so we need an additional preprocessing step to deal with this. However, first we need to show how to use non-simple paths together with the result of Corollary 11.

B.1 Using Non-Simple Paths

Let $\pi = (v_1, \ldots, v_\ell)$ be a non-simple path and let v be a vertex appearing at least twice on the path π, i.e., $v = v_i = v_j$ for some $i < j$. We say that the path π is *self-crossing* in v if the edges incident to v on π appear in the following circular order (v_{i-1}, v_i), (v_{j-1}, v_j), (v_i, v_{i+1}), (v_j, v_{j+1}) in the embedding. We say that a path π is self-crossing if π is self-crossing in some vertex v. Otherwise we say that π is *non-crossing*. If a vertex v appears at least twice on a non-crossing path π then we say that π *touches itself* in v. In Section 6, we assumed that the path π from f_s to f_t in G is simple. Now we will show that the weaker assumption that π is non-crossing suffices.

In order to work with non-crossing paths we will modify the graph G to make the path π simple. Let us sketch the idea: pick v as a vertex where π touches itself in G. Take i such that there is no other edge from π between edges (v_{i-1}, v_i), (v_i, v_{i+1}) in a cyclic order around v. Take all edges E_v incident to v that are embedded between and including the edges (v_{i-1}, v_i), (v_i, v_{i+1}). Now, add a new vertex v' to G and make the edges E_v to be incident to v' instead of v. Finally connect v with v' using an undirected edge of weight zero (see Figure 3). Starting with π a non-crossing path, perform this vertex-splitting operation until π becomes simple. This produces a new graph $G_{s,\pi}$. Note that this transformation does not change the weights of separating cycles, so we get the following observation.

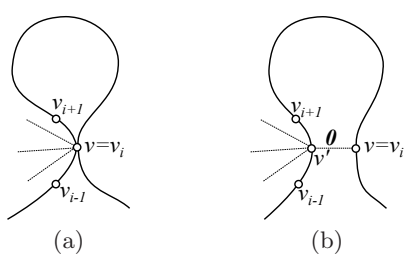

Figure 3: Dealing with non-simple paths. (a) A non-crossing path π of G that touches itself at vertex $v = v_i$. (b) The vertex splitting transformation at vertex v.

Corollary 16. *The weights of min st-separating cycles in G and $G_{s,\pi}$ are the same.*

As a result, if π is a non-simple non-crossing path, instead of running our algorithm on G, we can compute the graph $G_{s,\pi}$ and run our algorithm on it.

B.2 Additional Preprocessing

In the following, regard external faces of pieces as holes. For each hole h in a piece P, we fix a boundary vertex b_h. For each pair of holes h, h' in P, we fix a path $\pi_{h,h'}$ which is not self-crossing and which starts in b_h, ends in $b_{h'}$, goes through $b_{h''}$ for all holes h'' in P, and for all $b_{h''}$ on the path walks around the hole h'' passing through all its boundary vertices (see Figure 4). For each pair of holes h, h' in P we compute the dense distance graph in the piece obtained from P by cutting open along $\pi_{h,h'}$ and we find a min $b_h b_{h'}$-separating cycle $C_{h,h'}$ in P.

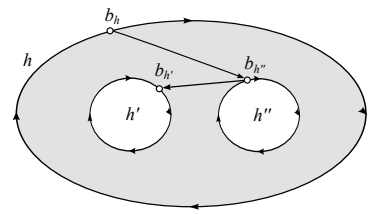

Figure 4: The path $\pi_{h,h'}$.

Lemma 17. *Additional processing takes $O(n \log r)$ time.*

Proof. For each piece P and each pair of holes the dense distance graph can be computed in $O(r \log r)$ time as in Lemma 2 and a min separating cycle can be found within the same time bound. Hence, over all pieces we need $O(n \log r)$ time. □

Now in order to connect f_s and f_t we will use paths $\pi_{h,h'}$ whenever we need to pass between two different holes h, h' in a piece P. Let $\mathcal{P}_\pi$ be the set of all such pieces on π. The resulting path is no longer simple, but it will be non-crossing. As shown in Appendix B.1, non-crossing paths can be dealt with as well. We make the following observation (see Figure 5).

Corollary 18. *A min st-separating cycle C either contains a vertex in $\partial\pi$ or is fully contained in a piece of $\mathcal{P}_\pi$.*

Proof. If cycle C contains a vertex in $\partial\pi$, we are done. Assume that it does not contain any vertex from $\partial\pi$. In order to be an st-separating cycle, it has to cross path π and it can do so in one of $\pi_{h,h'}$. Then it has to be fully contained in the corresponding piece P since by the construction of $\pi_{h,h'}$, all boundary vertices of P lie on π. □

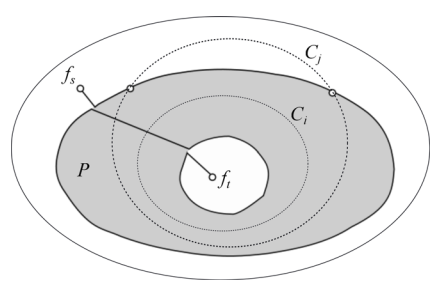

Figure 5: On illustrating Corollary 18. Either a separating cycle is fully contained in one of the pieces $\mathcal{P}_\pi$ (such as C_i) or it must contain one of the boundary vertices in $\partial\pi$ (such as C_j).

Using Corollary 18, we can find a min st-separating cycle in two phases. First, let C_i be the shortest $C_{h,h'}$ in $\mathcal{P}_\pi$ for $b_h, b'_h \in \pi$. Second, run the algorithm of Section 6.2 on a path π in $\overline{G}^2_{st}$ to find a cycle C_b. Finally, return the shortest of C_i and C_b. These computations do not increase the asymptotic running time of our dynamic algorithm.